



# SMART CONTRACT AUDIT REPORT

for

## Marlin POND Token



Prepared By: Xiaomi Huang

PeckShield

September 8, 2022

## Document Properties

Client	Marlin Protocol
Title	Smart Contract Audit Report
Target	POND Token
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author	Description
1.0	September 8, 2022	Luck Hu	Final Release
1.0-rc	September 05, 2022	Luck Hu	Release Candidate

## Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About POND Token . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	7
<b>2</b>	<b>Findings</b>	<b>8</b>
2.1	Summary . . . . .	8
2.2	Key Findings . . . . .	9
<b>3</b>	<b>ERC20 Compliance Checks</b>	<b>10</b>
<b>4</b>	<b>Detailed Results</b>	<b>13</b>
4.1	Excessive Contract Inheritance . . . . .	13
4.2	Trust Issue of Admin Keys . . . . .	14
<b>5</b>	<b>Conclusion</b>	<b>16</b>
	<b>References</b>	<b>17</b>

# 1 | Introduction

Given the opportunity to review the design document and related source code of the `Marlin` `POND` token contract, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract can be further improved due to the presence of certain issues related to ERC20-compliance, security, or performance. This document outlines our audit results.

## 1.1 About POND Token

`Marlin` is a secure platform-agnostic networking protocol that enables faster transmission of blocks and transactions between users and validators, making `web3` experiences smoother and cheaper. `POND` is the native staking token of the `Marlin` protocol and is used for helping to secure the network. The basic information of the audited token contract is as follows:

Table 1.1: Basic Information of POND Token

Item	Description
Name	Marlin Protocol
Website	<a href="https://www.marlin.pro/">https://www.marlin.pro/</a>
Type	ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	September 8, 2022

In the following, we show the Git repository of reviewed contracts and the commit hash value used in this audit. Note the audit only covers the `contracts/Pond.sol`.

- <https://github.com/marlinprotocol/Contracts.git> (ff8642a)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/marlinprotocol/Contracts.git> (c755730)

## 1.2 About PeckShield

PeckShield Inc. [6] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email ([contact@peckshield.com](mailto:contact@peckshield.com)).

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [5]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk;

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

We perform the audit according to the following procedures:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Table 1.3: The Full List of Check Items

Category	Check Item
<b>Basic Coding Bugs</b>	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead of Transfer
	Costly Loop
	(Unsafe) Use of Untrusted Libraries
	(Unsafe) Use of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
	Approve / TransferFrom Race Condition
<b>ERC20 Compliance Checks</b>	Compliance Checks (Section 3)
<b>Additional Recommendations</b>	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool does not identify any issue, the contract is considered safe

regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table [1.3](#).

## 1.4 Disclaimer

---



Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.



## 2 | Findings

### 2.1 Summary

Here is a summary of our findings after analyzing the `POND` token contract. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	0	
Informational	1	
Total	2	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.



## 2.2 Key Findings

Overall, no ERC20 compliance issue was found, and our detailed checklist can be found in Section 3. Also, though current smart contracts are well-designed and engineered, the implementation and deployment can be further improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 1 informational recommendation.

Table 2.1: Key POND Token Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	<a href="#">Excessive Contract Inheritance</a>	Coding Practices	Confirmed
PVE-002	Medium	<a href="#">Trust Issue of Admin Keys</a>	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 4 for details.



### 3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `view-only` Functions Defined in The ERC20 Specification

Item	Description	Status
<b>name()</b>	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
<b>symbol()</b>	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
<b>decimals()</b>	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
<b>totalSupply()</b>	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
<b>balanceOf()</b>	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
<b>allowance()</b>	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited POND Token. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-

Table 3.2: Key State-Changing Functions Defined in The ERC20 Specification

Item	Description	Status
<b>transfer()</b>	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
<b>transferFrom()</b>	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
<b>approve()</b>	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
<b>Transfer() event</b>	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to <i>address(0x0)</i> when new tokens are generated	✓
<b>Approval() event</b>	Is emitted on any successful call to approve()	✓

adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
<b>Deflationary</b>	Part of the tokens are burned or transferred as fee while on <code>transfer()/transferFrom()</code> calls	—
<b>Rebasing</b>	The <code>balanceOf()</code> function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
<b>Pausable</b>	The token contract allows the owner or privileged users to pause the token transfers and other operations	—
<b>Blacklistable</b>	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
<b>Mintable</b>	The token contract allows the owner or privileged users to mint tokens to a specific address	—
<b>Burnable</b>	The token contract allows the owner or privileged users to burn tokens of a specific address	—

## 4 | Detailed Results

### 4.1 Excessive Contract Inheritance

- ID: PVE-001
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: Pond
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

#### Description

The `POND` is an ERC20-compliant token which supports some extended features by integrating the `OpenZeppelin` library. Specifically, it inherits from the `AccessControlEnumerableUpgradeable` contract to support the role-based access control mechanisms, the `ERC20CappedUpgradeable` contract to support the supply capability, and the `UUPSUpgradeable` contract to support the upgradeability mechanism, etc. While reviewing these parent contracts which are inherited by the `Pond` contract, we notice the existence of excessive contract inheritance.

To elaborate, we show below the code snippet from the `Pond` contract. Specially, it inherits from both the `AccessControlUpgradeable` contract (line 21) and the `AccessControlEnumerableUpgradeable` contract (line 22). However, the `AccessControlEnumerableUpgradeable` contract also inherits from the `AccessControlUpgradeable` contract. So there is no need for the `Pond` contract to inherit directly from `AccessControlUpgradeable`. It only needs to directly inherit from the `AccessControlEnumerableUpgradeable` contract which further inherits from the `AccessControlUpgradeable` contract.

Similarly, we can remove the direct inheritance from `ERC20Upgradeable` (line 23) and `ERC1967UpgradeUpgradeable` (line 25).

```
17     contract Pond is
18         Initializable, // initializer
19         ContextUpgradeable, // _msgSender, _msgData
20         ERC165Upgradeable, // supportsInterface
21         AccessControlUpgradeable, // RBAC
22         AccessControlEnumerableUpgradeable, // RBAC enumeration
```

```

23     ERC20Upgradeable, // token
24     ERC20CappedUpgradeable, // supply cap
25     ERC1967UpgradeUpgradeable, // delegate slots, proxy admin, private upgrade
26     UUPSUpgradeable, // public upgrade
27     IArbToken // Arbitrum bridge support
28 {
29     // in case we add more contracts in the inheritance chain
30     uint256[500] private __gap0;
31     ...
32 }

```

Listing 4.1: Pond.sol

With this improvement on the inheritance graph, we can remove below function overriding of `_grantRole()`, as there is no function inheritance conflict nor new logic.

```

58     function _grantRole(bytes32 role, address account) internal virtual override(
        AccessControlUpgradeable, AccessControlEnumerableUpgradeable) {
59         super._grantRole(role, account);
60     }

```

Listing 4.2: Pond::\_grantRole()

**Recommendation** Remove the above mentioned redundant inheritance and remove the overriding of routine `_grantRole()`.

**Status** This issue has been confirmed by the team. And they decide to keep it as is to follow their best practice.

## 4.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Pond
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

### Description

In the Pond token contract, there is a privileged ADMIN account (assigned in the `initialize()` routine) that plays a critical role in governing and regulating the token-wide operations. To elaborate, we show below the sensitive operations that are related to the ADMIN role. Specifically, it has the authority to authorize the contract upgrade, grant/revoke BRIDGE\_ROLE member, mint/burn POND token for users, etc.

It would be worrisome if the ADMIN account is a plain EOA account. A multi-sig account could greatly alleviate this concern, though it is far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered for mitigation.

```

73     function _authorizeUpgrade(address /*account*/) internal view override {
74         require(hasRole(DEFAULT_ADMIN_ROLE, _msgSender()), "Pond: must be admin to
           upgrade");
75     }

```

Listing 4.3: Pond::\_authorizeUpgrade()

```

97     function setL1Address(address _l1Address) external onlyAdmin {
98         l1Address = _l1Address;
99     }
100
101     function bridgeMint(address _account, uint256 _amount) external onlyBridge {
102         _transfer(address(this), _account, _amount);
103     }
104
105     function bridgeBurn(address _account, uint256 _amount) external onlyBridge {
106         _transfer(_account, address(this), _amount);
107     }
108
109     function withdraw(uint256 _amount) external onlyAdmin {
110         _transfer(address(this), _msgSender(), _amount);
111     }

```

Listing 4.4: Pond.sol

**Recommendation** Promptly transfer the ADMIN privilege of POND token to the intended governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

**Status** This issue has been mitigated as the team confirmed that they need the admin capabilities for now to do the upgrade on the existing contract and set new parameters. These privileges will be renounced in future.

## 5 | Conclusion

In this security audit, we have examined the design and implementation of the `Marlin` `POND` token contract. `Marlin` is a secure platform-agnostic networking protocol that enables faster transmission of blocks and transactions between users and validators, making `web3` experiences smoother and cheaper. The audited `POND` token is the native staking token of the `Marlin` protocol and is used for helping to secure the network. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, although no critical or high level vulnerabilities were discovered, we identified two issues of varying severities. In the meantime, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.





## References

- [1] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-563: Assignment to Variable without Use. <https://cwe.mitre.org/data/definitions/563.html>.
- [3] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [4] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [5] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [6] PeckShield. PeckShield Inc. <https://www.peckshield.com>.